The ParaWise Expert Assistant – Widening Accessibility to Efficient and Scalable Tool Generated OpenMP Code*

Stephen Johnson, Emyr Evans, Constantinos Ierotheou
Parallel Processing Research Group, University of Greenwich,
Old Royal Naval College, Park Row, Greenwich, London SE10 9LS, UK
{s.johnson,e.w.evans,c.ierotheou}@gre.ac.uk

Haoqiang Jin
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
hjin@nas.nasa.gov

NAS Technical Report NAS-04-010, August 2004

Abstract

Despite the apparent simplicity of the OpenMP directive shared memory programming model and the sophisticated dependence analysis and code generation capabilities of the ParaWise/CAPO tools, experience shows that a level of expertise is required to produce efficient parallel code. In a real world application the investigation of a single loop in a generated parallel code can soon become an in-depth inspection of numerous dependencies in many routines. The additional understanding of dependencies is also needed to effectively interpret the information provided and supply the required feedback. The ParaWise Expert Assistant has been developed to automate this investigation and present questions to the user about, and in the context of, their application code. In this paper, we demonstrate that knowledge of dependence information and OpenMP are no longer essential to produce efficient parallel code with the Expert Assistant. It is hoped that this will enable a far wider audience to use the tools and subsequently, exploit the benefits of large parallel systems.

1 Introduction

The OpenMP standard [1] was devised to simplify the exploitation of parallel hardware, making it accessible to a far wider audience than most alternative techniques. Standardization has allowed a single application code to be portable to a wide range of parallel machines and compilers. The shared memory model followed by OpenMP allows simpler implementation than the explicit message passing model that has proved successful for

*The paper was prepared from a presentation given at the Workshop on OpenMP Applications and Tools (WOMPAT), Houston, Texas, May 2004.

distributed memory systems. Despite this, creating efficient, scalable OpenMP code still requires a significant degree of expertise as effective determination of, in particular, parallelism and variable privatization is essential. The overheads inherent in an OpenMP parallel region limit performance when inner loops in a nest operate in parallel, requiring outer loop parallelization in many cases to facilitate efficiency. This often adds the complexity of investigating interprocedural variable references to the process as loops containing call sites need to operate in parallel. Other key factors to efficient parallelization involve reduction of OpenMP overheads by having parallel regions containing many loops and avoiding unnecessary synchronization.

The necessary expertise and the volume of information that needs to be considered have limited the use and success of OpenMP parallelization. To ease these problems, automatic parallelizing compilers from vendors and research groups [2, 3, 18], and interactive parallelization tools [13, 17, 19] have been developed to detect parallelism based on some form of dependence analysis of the application.

Obviously, the ideal scenario for a user who wishes to parallelize their code is the automatic parallelizing compiler, however, in practice, the resultant performance is typically limited, particularly in terms of scalability. There are a number of reasons for this, with one fundamental problem being related to unknown input information such as the value ranges of variables that are read into the application code. This type of information can often be critical in accurate dependence determination and so the compiler is forced to conservatively assume the existence of a dependence, potentially preventing parallelism detection or valid privatization. Additionally, accurate analysis can be restricted by the time constraints a compiler must meet for commercial acceptability where in-depth analysis may be sacrificed to allow rapid operation.

The alternative approach is to use interactive parallelization tools to allow user involvement in the parallelization process while still exploiting the power of dependence analysis. The vital role of the user leads to the need for an accurate dependence analysis to only involve the user when necessary. A detailed interprocedural, value based dependence analysis can be used to produce a more accurate dependence graph, at the cost of a greater analysis time, but the consequent reduction in user effort is often deemed to warrant this expensive analysis. The Parawise/CAPO parallelization tools [4, 5, 6, 13] have been developed based on such a philosophy. Unfortunately, as the user is involved, some degree of expertise is again required to interpret information and provide the necessary feedback. This expertise may well include the understanding of the OpenMP parallelization strategy, including variable privatization, and also some level of understanding of dependence information. As a result, the tools may not provide increased accessibility to the production of efficient and scalable OpenMP code.

To overcome the limitations described above in the use of parallelization tools, we are developing an environment that will meet the needs of a wide range of potential users. For users with expertise in OpenMP parallelization, the clear presentation of complex interprocedural information and the automation of the vast majority of the parallelization

process are provided to greatly improve their productivity, allowing them to focus on fine tuning application codes for high parallel efficiency. For users with some experience with OpenMP, the environment is intended to relieve them of almost all parallelization decisions and avoid the trial-and-error approach to OpenMP parallelization often taken (e.g., where an assumption is made and tested, as a thorough interprocedural investigation of all the required issues proved overwhelming). Most importantly, the environment also aims to cater for users with no experience, and often no interest, in OpenMP parallelization. Such users typically have expertise in other areas of science and only require the computation power of parallel processing without much interest in how this is achieved. To make the environment accessible to these potential users, all aspects of OpenMP parallelization and dependence analysis must be kept away from the user, concentrating instead on information related to the application code being parallelized. If the necessary information required to enable efficient, scalable parallelization can be obtained from a user by only asking questions about their application code, then the number of users of tools could dramatically increase. In this paper we focus on the development of an Expert Assistant, a key component of the environment, that attempts to close the gap between what the tool needs to know to improve the parallelization and what the user knows about the application code.

2 ParaWise/CAPO OpenMP Code Generation

ParaWise is a parallelization tool from PSP Inc. [4, 6, 7, 12] that performs an in-depth interprocedural, value based dependence analysis of Fortran applications. It has been developed over many years and has numerous techniques to improve the quality of its analysis [4]. The CAPO module [13, 14] uses dependence information provided by ParaWise to generate effective OpenMP parallel code. The CAPO module includes many algorithms to enhance the quality of the generated code, including automatic merging of parallel regions in an interprocedural framework, routine duplication to allow differing levels of parallelism for different call sites, automatic insertion of NOWAIT directives to reduce synchronizations as well as parallelism detection and variable privatization determination.

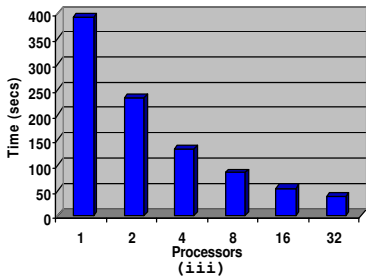
An example of the power of the CAPO OpenMP code generation is given by an example taken from an application code shown in Figure 1. In the original serial code (Figure 1(i)) there are two calls to routine `r2r`, one inside the `j` loop at S1, which is parallel, and the other outside that loop at S2. Inside routine `r2r` is another loop (`k`) that is also parallel. The ParaWise dependence analysis and CAPO algorithms automatically identify that these loops are parallel and determine that for the call site of `r2r` inside the `j` loop parallelism in that loop should be exploited, while for the call site at S2 parallelism in the contained `k` loop should be exploited (in this case, not considering multiple levels of parallelism in the `j` and `k` loops [14]). To allow this, routine `r2r` is automatically cloned

```
     do j=2,jmax
S1     call r2r(j,kmax)
     enddo
S2   call r2r(1,kmax)



   subroutine r2r(j,kmax)
   do k=1,kmax
     . . .
   enddo
```

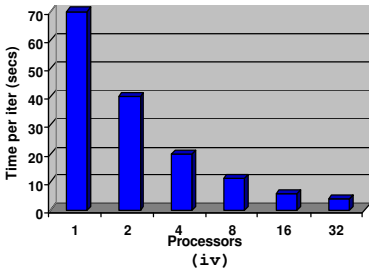
(i)

```
!$OMP PARALLEL DEFAULT(SHARED),PRIVATE(J)
!$OMP DO
      do j=2,jmax
        call cap_r2r(j,kmax)
      enddo
!$OMP ENDDO NOWAIT
      call r2r(1,kmax)
!$OMP END PARALLEL

      subroutine r2r(j,kmax)
!$OMP DO
       do k=1,kmax
         . . .
       enddo
 !$OMP ENDDO

       subroutine cap_r2r(j,kmax)
       do k=1,kmax
         . . .
       enddo
```

(ii)



(iii)



(iv)

Figure 1: Example of code generation techniques used in CAPO OpenMP code generation: (i) serial code, (ii) generated OpenMP code. Timings on Origin 2000 are shown for (iii) RJET code from Ohio Aerospace Institute, (iv) OVERFLOW from NASA Ames Research Center

(routine `cap_r2r`) and is referenced at S1 inside the `j` loop (Figure 1(ii)). The parallel region for the parallel `k` loop in the original routine `r2r` is then migrated to surround the call site at S2 and immediately follows the parallel region for the `j` loop. The parallel regions are then merged to form a single parallel region, reducing the region start and stop overheads incurred, and additional algorithms determine that there is no inter-thread interaction between the two loops so thread synchronization at the end of the first loop is not required and a NOWAIT can be generated. In this case, the interprocedural operation of the dependence analysis and code generation are essential for efficient parallel execution as hundreds of lines of code in the sub-call graph called within the `k` loop are executed in parallel.

Together, these tools can generate efficient, scalable OpenMP code and have been used successfully on many real world application codes [15, 16]. Figure 1(iii) shows timings for the RJET code from the Ohio Aerospace Institute that studies vortex dynamics and breakdown in turbulent jets, and also for the OVERFLOW code from the NASA Ames Research Center (Figure 1(iv)), used for aircraft simulations. Both codes show significant scalability up to, and beyond 32 processors.

Both ParaWise and the CAPO module have been designed with user interaction as a vital component, providing browsers to allow the user to inspect the parallelization and to add information about the application code. For OpenMP parallelizations, the Directive Browser shows loops that will operate in parallel and, more importantly, those loops that will execute in serial including an explanation of the inhibitors to parallelization. This

can involve variables that are assigned in an iteration of the loop and subsequently used in a later iteration and variables that are assigned or used in several iterations where, commonly, the assigned values appear to be used after the loop has completed. From this, the relevant dependencies can be investigated in another browser and, if possible, information added about variables in the application code that prove the dependencies do not exist (or alternatively the user can explicitly delete dependencies). In every real world application code that we have investigated some amount of user interaction is essential in the production of efficient and scalable parallel code. These tools can enable expert users of ParaWise/CAPO to produce efficient OpenMP code, however, an understanding of the application code, dependence analysis and OpenMP parallelization are all needed to make the most effective use of the tools. To reduce the effort and skill required by an expert, and allow non-experts to exploit the power of automatic parallelization techniques, the ParaWise Expert Assistant has been developed to identify the information that is required to enable effective parallelization and ask the user questions in the context of their application code.

3 The ParaWise OpenMP Expert Assistant

Ideally, parallelization experts who specialize in producing OpenMP parallel codes and code authors (or whoever wishes to parallelize an application code) should be able to use the ParaWise/CAPO tool. Our experience of parallelizing application codes clearly indicates that the information required to improve the efficiency of a parallel version will be known to the code author. The information required by the tool is the same information that would be needed by the parallelization experts undertaking a manual parallelization who currently need to ask the code author about certain variables in a code, although the techniques used to determine what information is essential are different.

The dependence information provides a rigid framework in which parallelism and privatization can be determined. Parallelism is determined by checking for loop-carried dependencies that enforce an execution order where one iteration of a loop must precede a later iteration of that loop. If the loop-carried dependence is a true (dataflow) dependence (i.e., an assignment followed by a later usage) then investigation of this dependence is required. If any loop-carried true dependence cannot be proven non-existent (except for being part of a reduction operation) then the loop must remain serial. If the loop-carried dependence is a pseudo dependence (memory location re-use, e.g., a usage followed by a later assignment, known as an anti dependence, or an assignment followed by a later re-assignment, known as an output dependence) then, to allow parallel execution either this dependence must be proven non-existent or all true dependencies of this variable into or out of the loop (e.g., an assignment in the loop to a usage following the loop) must be proven non-existent to enable the required privatization of this variable (except where FIRSTPRIVATE or LASTPRIVATE can be used).

For a pseudo dependence, the investigation only focuses on the memory locations of the references involved. For a true dependence, both the memory locations and any assignments in-between the assignment and usage of that dependence are investigated. The memory location test is often interprocedural as the assignment and usage can both be deep inside a call tree from call sites inside the loop. For the intermediate assignment test, we need to determine if all values passed from assignment to usage of the original dependence are overwritten (i.e., no value is passed). Again, this often involves interprocedural operation where the intermediate assignments can be in any number of different routines in the call tree. Additionally, for some loop-out dependencies, an interprocedural investigation in all possible combinations of caller sites may need to be made where other call sites that lead to usages of the subject variable must be considered. This leads to several types of question being asked by the Expert Assistant.

Firstly, conditions are presented to the user for inspection where the user can select and confirm any conditions as always TRUE. These conditions are extracted from the indices of arrays in the assignment and usage of the dependence under investigation, the control of these statements throughout the associated call paths and, in the case of true dependencies, indices and control for any intermediate assignments [4]. In most cases, this leads to fairly complex conditions for true dependencies which take the form:

$$Ar \cap Ac \cap Ur \cap Uc \cap (\neg((Ir_1 \cap Ic_1) \cup (Ir_2 \cap Ic_2) \cup ... \cup (Ir_n \cap Ic_n)))$$

where Ar is the range of the array that is assigned in the source of the dependence under investigation, Ac represents the conditions under which that assignment occurs, Ur and Uc are the index range and control for the usage at the sink of the dependence, and Ir_j and Ic_j are the index range and control for each intermediate assignment identified.

These conditions are extracted from the heart of the ParaWise dependence analysis to encourage substitution of variables to those read into the application code (as these are often more familiar to a user than variables calculated as part of an algorithm in the application) and also to automatically prove many cases so that the user only sees cases that need their attention. The algorithms in the dependence analysis substitute sets of constraints through conditional assignments and multiple call sites to produce a set of fully substituted constraints. If a set is presented to the user and they can confirm some constraint then that set is proven impossible and the next unresolved set is processed. If the user cannot provide information for a set then that set remains unresolved and the overall test terminates, indicating that the dependence concerned cannot be proven non-existent. Secondly, true dependencies where intermediate assignments exist are presented to the user, both graphically and in a list form, showing all intermediate assignments in the context of the application code. Such information can enable the user to understand what is being asked and, with their understanding of the algorithm implemented in the code section concerned, determine if the dependence in question carries any values from its source to its sink.

The Expert Assistant also identifies other situations, such as when an I/O statement is inside a loop (searching in call sites within the loop), and asks whether it is essential to the operation of the code or just for debugging purposes.

Questions are asked either for a selected individual loop or for all loops listed in the Directive Browser. Once a question and answer session is complete, the added information and deletions of dependencies or I/O statements etc. are applied and the parallelization process is automatically replayed. In some cases, the dependence analysis will need to be repeated where a replay file is created containing all actions in the previous parallel process with the new actions also added. In other cases, the previously constructed dependence graph can be altered to include the effect of the new information.

4 Examples of the Power of the Expert Assistant

4.1 Simple constraints needed to allow privatization

A common cause of serialization is when a workspace variable that needs to be privatized to allow parallelization (due to a loop-carried pseudo dependence) cannot be privatized due to usages after the loop has completed. An example of this is shown in Figure 2(i) where the loop `i1` writes to array `b` (amongst many others) in several iterations causing an output dependence for instances of statement S1 between iterations of loop `i1`. An investigation of this dependence using the Expert Assistant reveals that no possible solutions exist that would allow its deletion so the process then determines if privatization is possible. Here, it is not possible as usages of the variable are made after the loop has completed, including the usage at S3 shown in Figure 2(i) (assuming that other code not shown prevents a LASTPRIVATE clause being used for `b`). The investigation therefore focuses on the causes of these post loop usages of values of array `b`. This can involve interprocedurally tracing many chains of dependencies relating to routine starts, routine stops, call sites and also the actual statements containing usages (i.e., in Figure 2(i), loop `i2` will often reside in another routine).

The investigation of the dependence from S1 to S3 uses the ParaWise dependence analysis engine test for value based dependencies, identifying intermediate assignments of the values between S1 and S3. In this case, statement S2 is included and the resultant constraints lead to the question "Is `nj` always ≥ 2?". The variable `nj` is read at runtime and should be familiar to a user who is knowledgeable about the application code. Typically, the operation of the code makes no sense if such constraints are not met (i.e., the code was written making the implicit assumption that `nj` will be ≥ 2) allowing the user to answer with confidence. Additionally, since parallel processing is only needed for computationally expensive executions of the application code, the user may be able to indicate that the value of `nj` will always be at least 2 for a use of the parallel version. In Figure 2(i), the assignment to `b` at S2 only occurs if `nj` ≥ 2 (otherwise the surrounding `j2` loop is

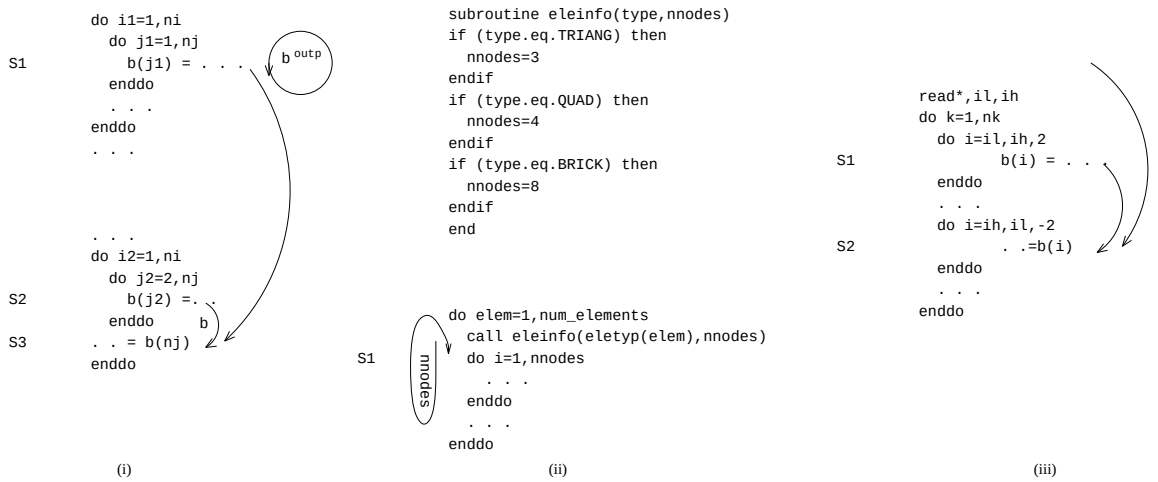


Figure 2: Simple case of (i) a privatization preventing post-loop dependence, (ii) code fragment where element type is related to a number of nodes for each element, and (iii) potentially covering assignment for the usage under certain conditions

not entered). If `nj` were less than 2 then the values used at S3 would come from other assignments, potentially taking values from throughout the application code. In this case, as in all the other cases highlighted in this section, it is not a deficiency in the analysis that has caused this dependence, the conservative nature of dependence analysis and lack of information about variable `nj` forces it to be set. Also, the reason that array `b` in loop `i1` could not be privatized was not related to loop `i1`. The automatic investigation of the Expert Assistant traced the problem to loop `i2` elsewhere in the application code. By asking a question about the constraint on a variable, the user never needs to know where the problem was traced to. This avoids one of the major tasks in the use of previous versions of CAPO. The addition of knowledge in this way means that the information could prove crucial in proving that other dependencies can be removed.

4.2 More complex cases for a finite element code

A finite element application code provides further examples of the power of the Expert Assistant. This code follows the fairly standard finite element algorithm where a number of different element types are used to form the mesh. Stress and strain components are calculated for the nodes of each element and are based on shape functions and boundary conditions. The code provides three types of finite elements: 2D triangular elements (3 nodes), 2D quadrilateral elements (4 nodes) and 3D brick elements (8 nodes). It has four modes of operation: plain stress, plain strain, axis symmetric and full three-dimensional analysis. As is fairly typical with such codes, the mesh is read in from a file that has been produced by a mesh generation package, therefore it will conform to certain constraints. In this case, these include that the elements are one of the three types above, that the

analysis mode is one of four above, and that brick elements are only used with the three-dimensional analysis mode. As these constraints are always met by meshes read from file, they were not checked in the application code, and therefore this information can only be obtained from the user.

A simple case that was caused by the implicit constraint on finite element types is shown in Figure 2(ii). The variable `nnodes` is assigned for all valid element types of the code, however, as the array `eletyp` is read from a file and no explicit constraints are in the code, the analysis must assume that `eletyp` can take any value, and not necessarily one of the three required. As a result of the uncertainty of the assignment of variable `nnodes`, dependencies for the usage of `nnodes` at S1 are set to assignments of `nnodes` elsewhere in the application code, including dependencies to assignments in earlier iterations of the `elem` loop over finite elements, serializing that loop. The Expert Assistant automatically investigates the serializing dependence shown in Figure 2(ii) where the assignments to `nnodes` in the same iteration of the `elem` loop as the usage are intermediate assignments, so their control is collected for the questions to be asked of the user. The Expert Assistant interface lists all three conditions, along with a number of other conditions, where if any single condition is known to be TRUE the dependence does not exist. In this case, no single condition is TRUE, however, the interface allows for the combination of constraints. To provide the necessary information, the user selects the constraints and combines them to produce:

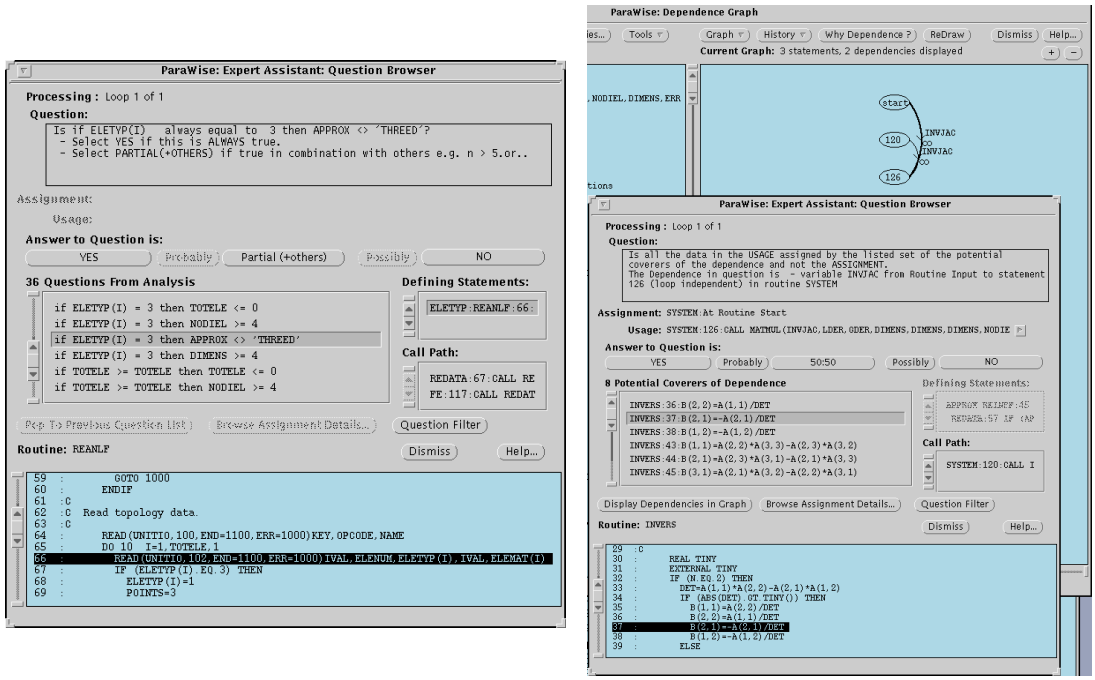
```
(eletyp(elem).eq.TRIANG.or.eletyp(elem).eq.QUAD.or.eletyp(elem).eq.BRICK)
```

where the user knows that this is always true since `eletyp` is read from the input file. This information allows the replayed dependence analysis to determine that such dependencies do not exist as the value of `nnodes` used at S1 in Figure 2(ii) is definitely from one of the three assignments shown earlier in the same iteration of loop `elem`. This may then allow parallelism in loop `elem` to be detected.

The form of presentation of the required constraint, and the variables involved, can assist user recognition and enable a positive response. An example of this is when two constraints are identified for a loop serializing dependence where an if-then presentation is more appropriate to the users understanding of the application code, i.e., the following facts are identical:

```
"(eletyp(elem).ne.TRIANG.or.mode.ne.threed)"
"if (eletyp(elem).eq.TRIANG) then (mode.ne.threed)"
```

where the user can far more easily understand the constraint when it states that if a triangular element is used then the analysis mode cannot be three-dimensional. An example of the Expert Assistant window for this scenario is shown in Figure 3(i). For the user to interpret such questions it requires the variables involved to be familiar to the user. Additionally, for the information to be applicable to many situations throughout the application code, the information should, as far as is possible, not relate to locally



(i) (ii)

Figure 3: Examples of the Expert Assistant window: (i) asking for confirmation of constraints (ii)asking if the set of intermediate assignments provides all values to a usage.

computed variables. In Figure 2(ii), for example, it may be difficult for a user to answer questions involving variable `nnodes` as it is computed locally. If the substitutions of the possible values of `nnodes` are exploited through conditional substitution then `nnodes` is eliminated from any questions and variable `eletyp` is introduced. As `eletyp` is read into the application code, its values are fundamental to the finite element method used and should be understandable to the user, and since this variable is used throughout the application code, any added information can potentially be exploited to resolve many other situations.

There are also cases in the finite element code where dependencies are inhibiting parallelism but no answerable question is presented to the user. The main cause of this is error checking, particularly in the finite element stiffness matrix calculation phase of the code. As part of this process, a small matrix is inverted so a check is used to avoid divisions by zero when the matrix is singular. This check can have a significant effect on the parallelization. Firstly, it may contain I/O and execution termination if a singular matrix is encountered. Secondly, if execution is not stopped then the possibility of data being used from the previously processed finite element (as it is not assigned for the element with the singular matrix) forces the serialization of the finite element loop. The Expert Assistant deals with such cases by detecting the I/O and program termination statements when they are inside a potentially parallel loop (in this case in a routine called

from inside such a loop) and presenting the statement in the context of the application code, asking the user if it is essential to the operation of the application code, or was just for debugging purposes during development. In this case, the generated finite element mesh read into the application code guarantees that no finite element will have a singular matrix so the user can recognize that these statements are just for debugging and can be removed for a production version of the parallel code.

The lack of an assignment of the inverted matrix still leads to serializing dependencies as the analysis does not know that no singular matrices will be encountered. As the detection of a singular matrix is based on a non-zero value of the matrix determinant, the questions posed by the Expert Assistant involving constraints cannot be answered by the user. Instead, the associated dependence is presented to the user in both graphical and textual form as shown in Figure 3(ii). The dependence that is causing the problem is highlighted in bold and the intermediate assignments that could overwrite all values carried by that dependence are also shown. Several other browsers are provided to offer more information about what is being asked (and all other ParaWise browsers are also available) where all these browsers show the statements involved in the context of the application source code. With their knowledge of the algorithms implemented in the application code, the user should be able to know that all values of the inverted matrix used for a finite element are calculated earlier in the same iteration of the finite element loop. This should allow them to answer with confidence that the dependence in question does not carry any values and can therefore be ignored in parallelism determination.

4.3 Other essential user interaction

Implicit constraints and error checking are often the cause of serialization in application codes, however, other scenarios also exist that can only be addressed by the user. Consider the example shown in Figure 2(iii) where a usage at S2 of values assigned before the code fragment shown could prevent privatization and therefore parallelism in other loops in the code. The values used at S2 could also have been provided by the assignment at S1, however, the loop step indicates that the values of the loop bounds `il` and `ih` as read into the application code are vital in determining which dependence exists. If the user knows that `il` and `ih` are both even, then the even indices of array `b` assigned at S1 will be used at S2 in the every iteration of loop `k`. Similarly, if both `il` and `ih` are odd then the odd indices of array `b` will be assigned and used in each `k` iteration. If however, `il` is even and `ih` is odd (or vice-versa) then no value used at S2 is assigned at S1 so the dependence from before this code fragment does exists (and the dependence from S1 to S2 does not exist). If no information is known by the user about `il` and `ih`, the worst case scenario must be taken to ensure conservative analysis, with both dependencies being set. In this case, the graphical and textual displays of the application code shown in Figure 3(ii) should enable the user to interpret and, if possible, resolve such situations.

4.4 Experience of using the Expert Assistant

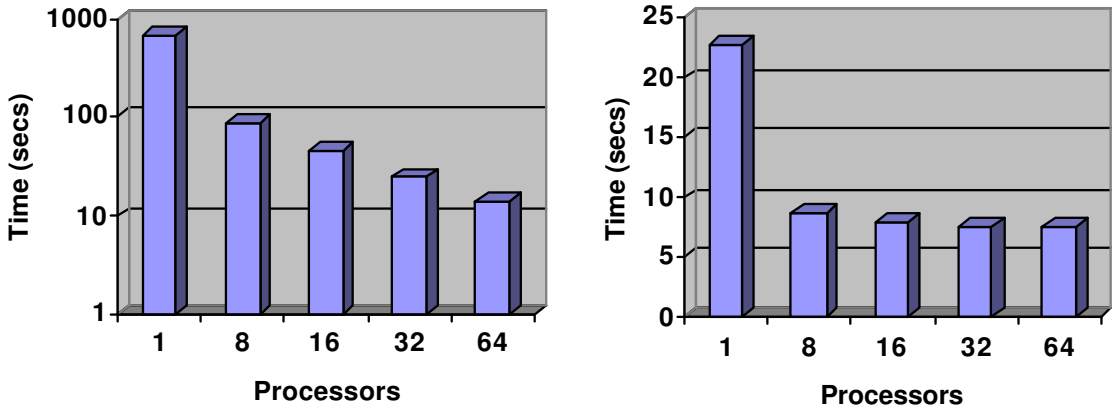
We have used the Expert Assistant on a large number of application codes and found that it does address the issues a parallelization expert would need to investigate to improve the parallelization quality, but does so in terms that should be understandable to a code author.

Results for parallelizations using the Expert Assistant are shown in Figure 4 for (i) the NAS benchmark BT and (ii) the Finite Element code discussed in section 4.2. For BT, the initial performance, when no user information was added, was very poor due to inner loops operating in parallel, so an investigation using the Expert Assistant was required. The questions asked by the Expert Assistant focused on constraints for the mesh size variables NX, NY and NZ where lower bounds to their values were of interest. Confirming that such constraints were always true for an execution of BT and performing a replay produced a parallel version with far superior performance, exhibiting high speedup and scalability to reasonably large numbers of processors as shown in Figure 4(i). In this version the outermost parallel loops were exploited in most cases.

For the Finite Element code, the initial parallelization with no user information was forced to focus mainly on inner loops due to dependencies between iterations of outer loops. After a session with the Expert Assistant providing the sort of information discussed in section 4.2, parallelism at outer loops was detected enabling some speedup and scalability to be achieved as shown in Figure 4(ii). A significant number of loop-carried dependencies and dependencies into and out of loops were proven non-existent in the replay allowing effective parallelism and privatization directives to be inserted. This version was able to exploit parallelism between iterations of the loop over finite elements, attaining the most effective level of parallelization possible in those cases. Although there is a marginal increase in performance from 8 to 32 processors, scalability is still limited. The main cause of this is that one serial code section (containing a single loop) still exists in the parallel code where, although this section seems relatively unimportant on a single processor, when larger numbers of processors are used, the speedup of parallel loops leads to this serial loop dominating runtime.

5 The Evolving Parallelization Environment

The Expert Assistant together with ParaWise and CAPO are powerful tools for parallelization, however, they are just part of the overall environment that is under development. The environment consists of a number of interrelated components, as shown in Figure 5, aiming at both OpenMP and message passing parallelizations. The exploitation of user knowledge is vital for effective parallelization, however, this inevitably opens the parallelization process to the danger of incorrect user assertions and resultant incorrect parallel execution. To address this inevitable problem in the environment, another component, based on relative debugging [8], is being developed to automatically identify where



(i) (ii)

Figure 4: Timing of (i) BT using log scale and (ii) the Finite Element code on 1-64 400MHz processors of an SGI Origin 3000

incorrect parallel calculations originate and to indicate the related potentially incorrect user decisions identified in the parallelization history. The relative debugger contains many features including tolerance settings for absolute and relative value comparisons and visualization of differences across threads or processors.

The final component is the parallel profiler and trace visualizer. A number of these tools already exist including Paraver from CEPBA [9], the SUN Studio 9 Performance Analyzer [10] and Intel VTune [11]. The aim in this work is to extract simple metrics (loop speedup etc.) to enable inexperienced users to focus their effort as well as allowing skilled users to exploit the full power of these tools with the additional information provided by the other environment components.

6 Related Work

It is well known that there is a need for information relating to the application code when performing a parallelization that is not statically available. Hence the attempts to address this issue range from runtime testing to fully manual parallelization decisions (such as those manually written with OpenMP).

A number of tools have been developed to assist users in shared memory directive based parallelization and only a subset are cited here [17, 18, 19, 20]. Most focus on providing information to the user, relying on the user to have the skill to correctly interpret and exploit the provided information that is often detailed. For parallelization experts, such tools provide a means to overcome some of the most difficult problems in code parallelizations relating to understanding the application code and investigating interprocedural accesses of data which can be significant manual tasks.

A number of techniques have been developed that involve checking for valid parallel

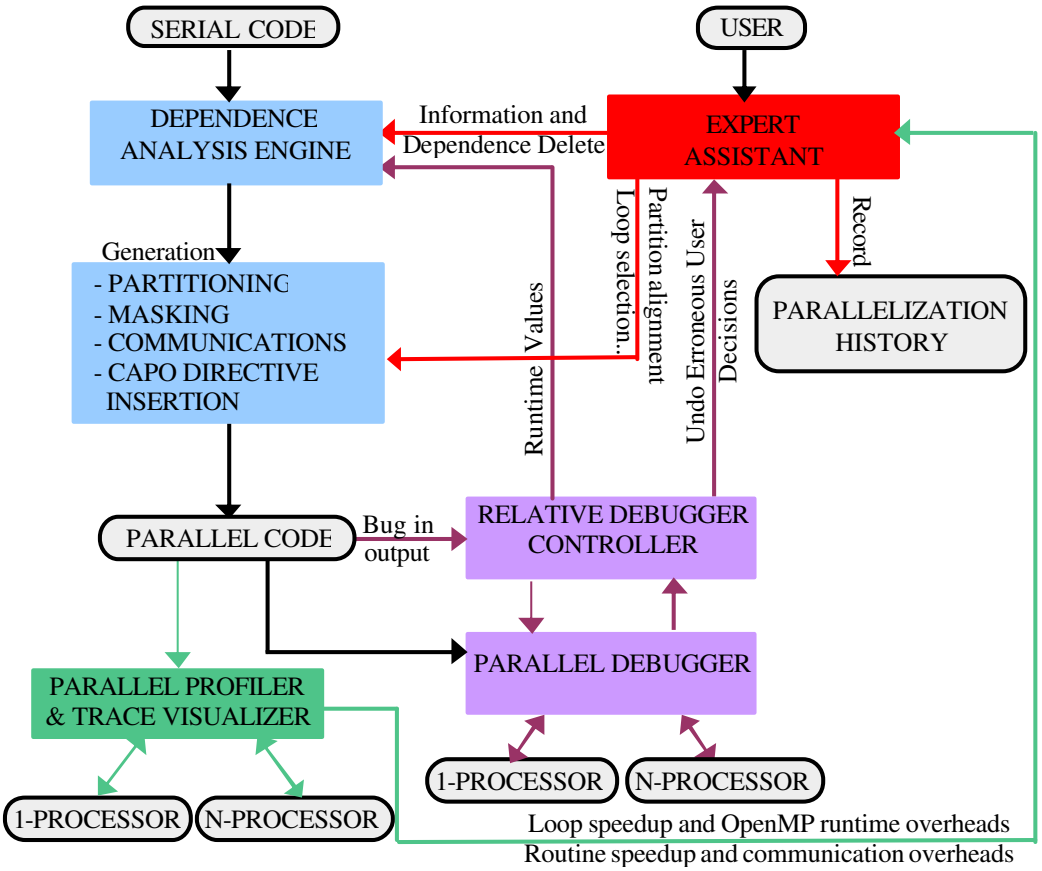


Figure 5: Components of the evolving parallelization environment

execution at runtime. The inspector/executor technique has been used to determine the legality of parallel execution [21]. The inspector loop, which is a light-weight duplicate of the computations that mimic the data accesses in the loop, uses these accesses to determine if parallel execution is legal. Both parallel and serial versions of the loop are generated and the appropriate version is executed after the test is made. Speculative computation [22] is where a loop is executed in parallel, but the references that potentially cause serializing dependencies are monitored. As soon as a serializing dependence is detected, the speculative parallel execution can be terminated and the serial version is then used. Some form of checkpointing is required to preserve the state of associated program data before the speculative parallel loop is executed to enable that state to be reset if serial execution is forced. Both these methods require an accurate, value based, interprocedural dependence analysis as a pre-requisite to only require such methods to be used in certain loops and to keep the number of variables whose accesses are monitored in those loops and the associated runtime overheads as low as possible. In addition, the prevention of privatization by, in particular, loop-out dependencies where a LASTPRIVATE directive cannot be used, will not be overcome using these methods as the runtime detection of output and anti dependencies for these shared variables will force serial execution. These

runtime techniques are focused on an automatic compiler approach to parallelization with little or no user interaction, so any serial code sections encountered will greatly restrict scalability. Presentation of such cases to the user is still essential if scalability is required.

7 Conclusion

The Expert Assistant achieves its goal of greatly simplifying the users role in the parallelization of real world application codes with the ParaWise/CAPO tools. Our experience demonstrates that the potential of Expert Assistant would be a major benefit to experienced users of the tool and we are confident that parallelization experts and those familiar with OpenMP will also find it invaluable. An important issue that remains to be answered is whether the Expert Assistant can encourage and enable the vast number of novice users to use the tools so they can take advantage of parallel machines with many processors.

Improvements that are currently underway include ordering questions based on the likelihood that they can be answered and will be profitable, and removing questions where a related variable is indicated as unanswerable by the user.

Future versions will be more closely coupled with CAPO algorithms to allow investigation of other cases such as where a NOWAIT could not be used or where ParaWise code transformations are advantageous. Additionally, the information from the profiling and relative debugging components of the environment will be used to identify crucial cases for user investigation.

Acknowledgements

The authors would like to thank their colleagues that have assisted in the many different aspects of this work, including Gabriele Jost, Greg Matthews and Bob Hood (NASA Ames), Peter Leggett and Jacqueline Rodrigues (Greenwich). This work was partly supported by the AMTI subcontract No. Sk-03N-02 and NASA contract DDTS59-99-D-99437/A61812D.

References

[1] OpenMP home page, http://www.openmp.org/.

[2] Wilson R.P., French R.S., Wilson C.S., Amarasinghe S.P., Anderson J.M., Tjiang S.W.K., Liao S., Tseng C., Hall M.W., Lam M. and Hennessy J., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," Stanford University, CA, 1996.

[3] Blume W., Eigenmann R., Fagin K., Grout J., Lee J., Lawrence T., Hoeflinger J., Padua D., Tu P., Weatherford S., "Restructuring Programs for High Speed Computers with Polaris," ICPP Workshop on Challenges for Parallel Processing, pp149-162, 1996.

[4] Johnson S.P.,Cross M., Everett M.G., "Exploitation of Symbolic Information in Interprocedural Dependence Analysis," Parallel Computing, 22, pp 197-226, 1996.

[5] Leggett P.F.,Marsh A.T.J., Johnson S.P. and Cross M., "Integrating user Knowledge with Information from parallelisation Tools to Facilitate the Automatic Generation of Efficient Parallel FORTRAN code.," Parallel Comp., Vol 22, 2, pp197-226, 1996.

[6] Evans E.W., Johnson S.P., Leggett P.F. and Cross M., "Automatic and Effective Multi-dimensional Parallelisation of Structured Mesh Based Codes," Parallel Computing 26(6): 677- 703, 2000.

[7] Johnson S.P., Ierotheou C.S. and Cross M., "Computer Aided Parallelisation of Unstructured Mesh Codes," Proceedings of PDPTA Conference, Las Vegas, volume 1, CSREA, pp 344-353, 1997.

[8] Matthews G., Hood R., Jin H., Johnson S. and Ierotheou C., "Automatic Relative Debugging of OpenMP Programs," Proceedings of EWOMP, Aachen, Germany 2003.

[9] Paraver, http://www.cepba.upc.es/paraver/.

[10] SUN Studio performance analyzer, http://developers.sun.com/prodtech/cc/analyzer_index.html.

[11] Vtune Performance Analyzer, http://developer.intel.com/software/products/vtune/index.htm.

[12] Parallel Software Products Inc, http://www.parallelsp.com/.

[13] Jin H., Frumkin M., Yan J., "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," Workshop on OpenMP Experience and Implmentation (part of International symposium on High Performance Computing), Tokyo, Japan p440, 2000; LNCS 1940, pp 440-456.

[14] Jin H., Jost G., Yan J., Ayguade E., Gonzalez M. and Martorell X., "Automatic Multilevel Parallelization Using OpenMP," Proceeding of EWOMP 2001, Barcelona, Spain, September 2001; Scientific Programming, Vol. 11, No. 2, pp 177-190, 2003.

[15] Ierotheou C.S., Johnson S.P., Leggett P., Cross M., Evans E., "The Automatic Parallelization of Scientific Application Codes Using a Computer Aided Parallelization Toolkit," Proceedings of WOMPAT 2000, San Diego, USA, July 2000; Scientific Programming Journal, Vol. 9, No. 2+3, pp 163-173, 2003.

[16] Jin H., Jost G.,Johnson D., Tao W., "Experience on the Parallelization of a Cloud Modelling Code Using Computer Aided Tools," NAS Technical Report NAS-03-006, NASA Ames Research Center, March 2003.

[17] FORGE, Applied Parallel Research, Placerville California 95667, USA, 1999.

[18] KAI/Intel, http://www.kai.com/.

[19] Zima H.P., Bast H-J. and Gerndt H.M., "SUPERB-A Tool for Semi-Automatic MIMD/SIMD Parallelisation," Parallel Computing, 6, 1988.

[20] The Dragon Analysis Tool, http//www.cs.uh.edu/~dragon/.

[21] Rauchwerger L., Amato N. and Padua D., "A Scalable Method for Run-Time Loop Parallelization," International Journal of Parallel Processing, vol 26, no. 6, pp537-576, July 1995.

[22] Rauchwerger L. and Padua D., "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," IEEE Transactions on Parallel and Distributed Systems, Vol 19, No 2, February 1999.